コンピュータアニメーション特論

第7回　衝突判定・ピッキング

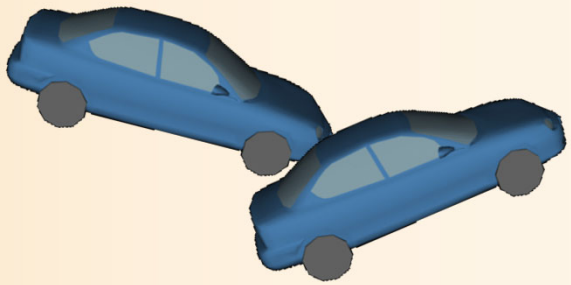
九州工業大学 情報工学研究院　尾下真樹

今日の内容

- 衝突判定
  - 近似形状による衝突判定
  - 空間インデックス
  - ポリゴンモデル同士の衝突判定
- ピッキング
  - サンプルプログラム
  - スクリーン座標系での判定
  - ワールド座標系での判定
  - レポート課題

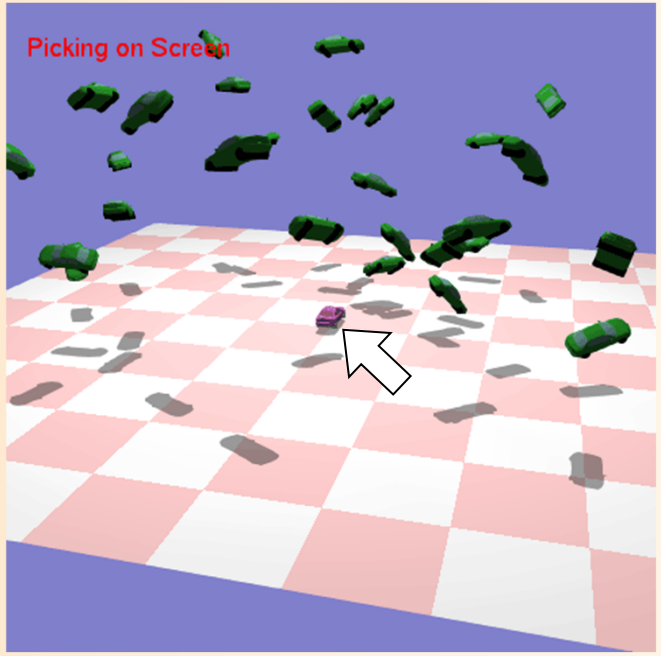
衝突判定

- ３次元空間において物体同士の衝突（交差）の有無や衝突箇所を判定する技術
  - 物理シミュレーション等、物体間の相互作用に応じてアニメーションを生成するためには、必須となる
  - 幾何学的な計算が必要
  - 高速化のための工夫も必要



ピッキング

- 画面上で３次元空間の物体を選択する操作
  - 画面上の指定した点に表示されているオブジェクトを判定する技術
  - 多くのアプリケーションで必要となる技術
  - 場合によっては、選択点の座標まで求める必要がある
  - 幾何学的な計算が必要



衝突判定

衝突判定の必要性

- アニメーションやシミュレーションにおいて、物体同士の衝突や接触を扱うためには、
  - どの物体同士が接触しているか
  - 接触している場合、どの部分が接触しているかを、毎ステップごとに判定・計算する必要がある
- 高速化の必要性
  - 任意の物体（ポリゴンモデル）同士が接触しているかどうかの判定には、時間がかかる
  - 特に空間に大量の物体が存在する場合、全ての組み合わせを判定していると時間がかかる

衝突判定（交差判定）の各種技術

- 近似形状による衝突（交差）判定
- 空間インデックス
- ポリゴンモデル同士の衝突（交差）判定

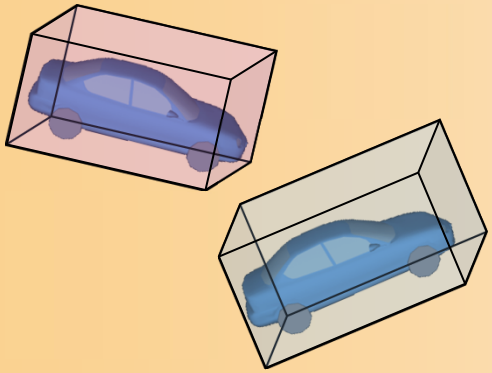
- 「衝突判定」 or 「交差判定」 の用語の区別
  - 同じ意味を表す
  - 物理シミュレーション的には「衝突」だが、幾何学的には３次元空間における物体の領域同士の「交差」と見なせるため、両表現が用いられる

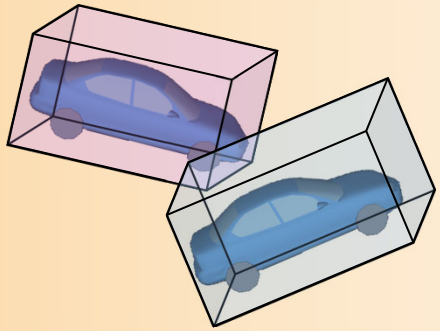
衝突判定

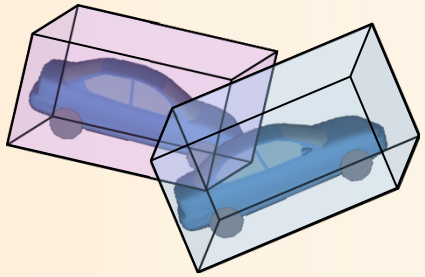
- 近似形状による衝突（交差）判定
- 空間インデックス
- ポリゴンモデル同士の衝突（交差）判定

近似形状による衝突判定

- 物体を内包する近似形状同士で判定を行う
  - 近似形状同士が交差しなければ、物体同士も交差しない
  - 効率的に交差しないことが判定ができる
- 近似形状同士が交差するときにのみ、もとの形状による正確な交差判定を行う







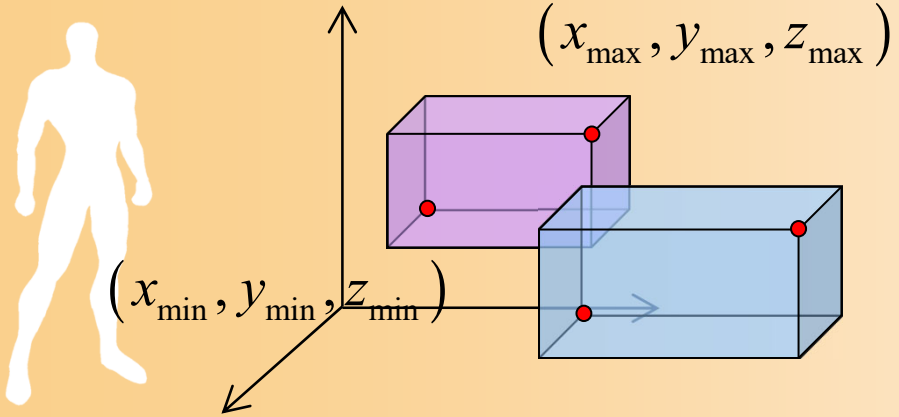
近似形状の種類

- 一般的に用いられる近似形状
  - 軸に平行な直方体（Axis Aligned Box）
  - 任意方向の直方体（Oriented Box）
  - 球（Sphere）

- 物体の形状や回転の有無によって、最適な近似形状は異なる

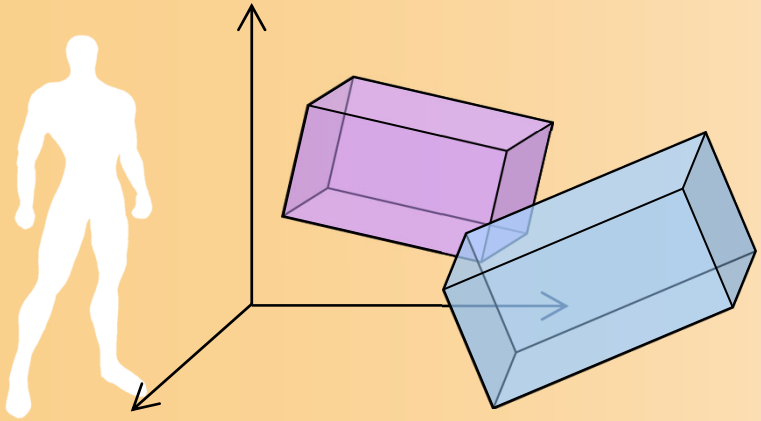
近似形状

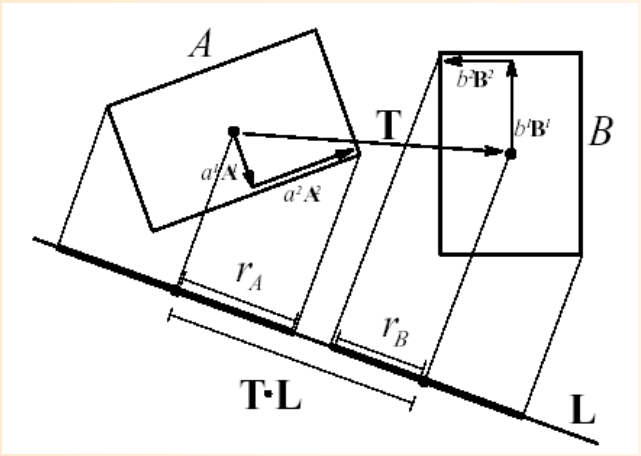
- 軸に平行な直方体（Axis Aligned Box）
  - 表現が簡単、判定が高速
    - 全ての軸で両者の範囲が重なるかどうかを判定
  - 物体が回転すると、直方体の大きさを再計算する必要がある
  - 無駄な空間ができるため、精度は低くなる



近似形状

- 任意方向の直方体（Oriented Box）
  - 物体を直方体で近似し、物体に合わせて回転
  - 判定にやや時間がかかる
    - 軸に投影したときの重なり（中心同士の距離）から判定
    - 最大15本の軸で判定
      - 3面×2個の法線＋3辺×3辺
    - 全ての軸上で重なれば交差

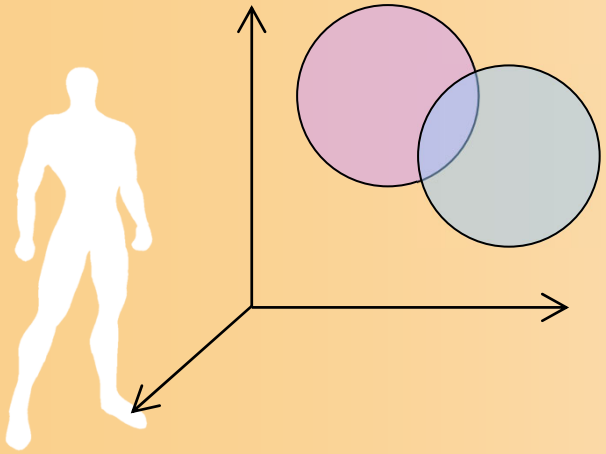


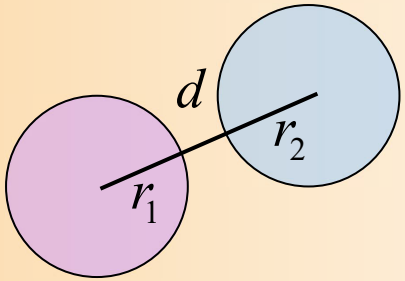


Gottschalk et al., “OBBTree: A Hierarchical Structure for Rapid Interference Detection”, SIGGRAPH ’96.

近似形状

- 球（Sphere）
  - 表現が簡単、判定も高速
    - 中心の距離と、各球の半径から、交差判定ができる
  - 回転しても近似形状の更新は不要
  - 物体の形状によっては、精度がかなり低くなる





衝突判定

- 近似形状による衝突（交差）判定
- 空間インデックス
- ポリゴンモデル同士の衝突（交差）判定

空間インデックス

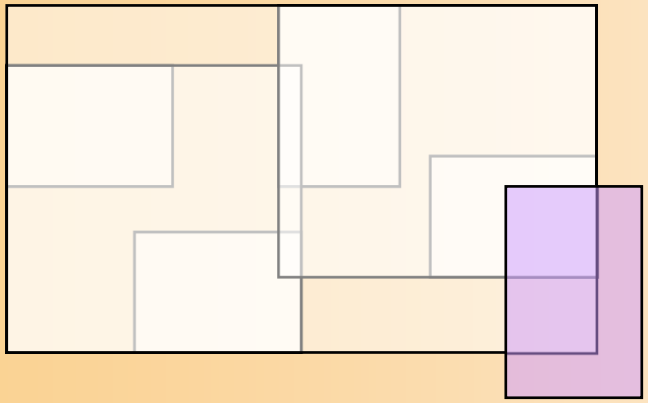
- 大量の物体がある場合、近似形状を用いたとしても、全物体同士の交差判定を行うと非常に時間がかかってしまう
- 交差する可能性がある物体同士のみ、判定を行いたい
- ある物体と交差する可能性がある物体を求めるために、空間インデックスを利用
  - 空間内のどの範囲にどの物体が存在するかを空間インデックスにより管理

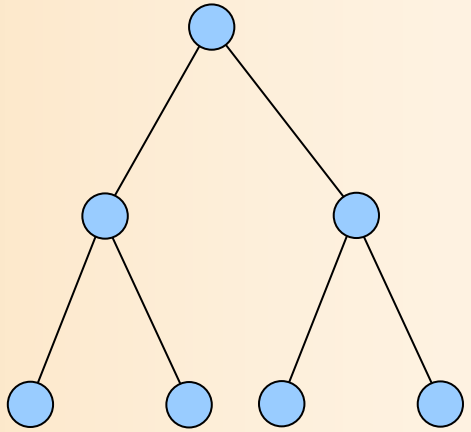
空間インデックスの種類

- ツリー
  - 近似形状の階層化によるツリー
  - オクトツリー
  - 空間２分割木（Binary Space Partitioning）
- ボクセル

近似形状の階層化によるツリー

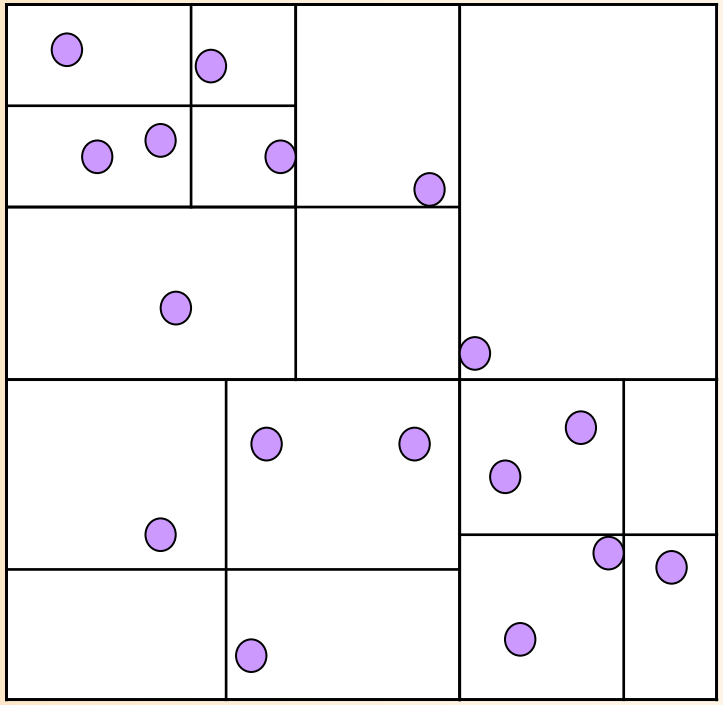
- 空間内の全物体の近似形状を階層化
  - 複数の近似形状を内包する、上位ノードの近似形状を階層的に生成
    - 物体が移動・回転する度に更新する必要がある
  - 上位ノードの近似形状と交差しなければ、下位ノードとの判定は省略できる





オクトツリー

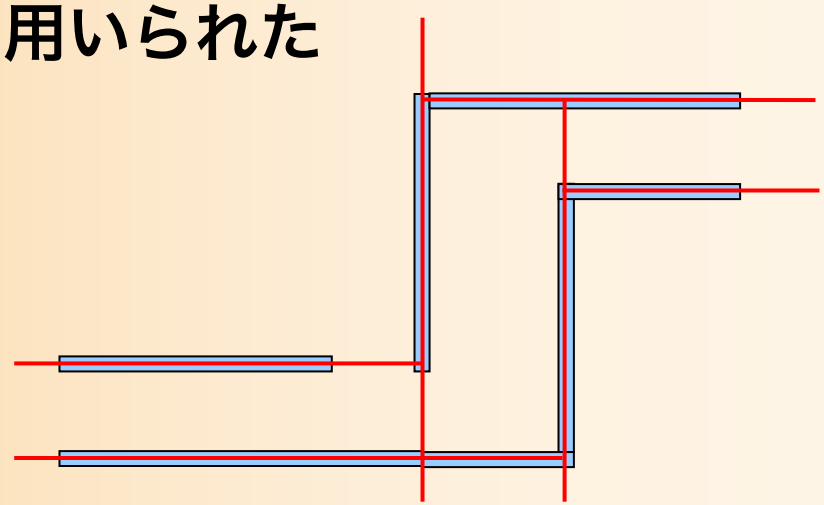
- 空間を階層的に８分割
  - 各領域ごとにどのオブジェクトが存在するかを記録
  - データ構造が単純
  - 更新に時間がかかる



２次元の例
（2次元の場合は4分割になる）

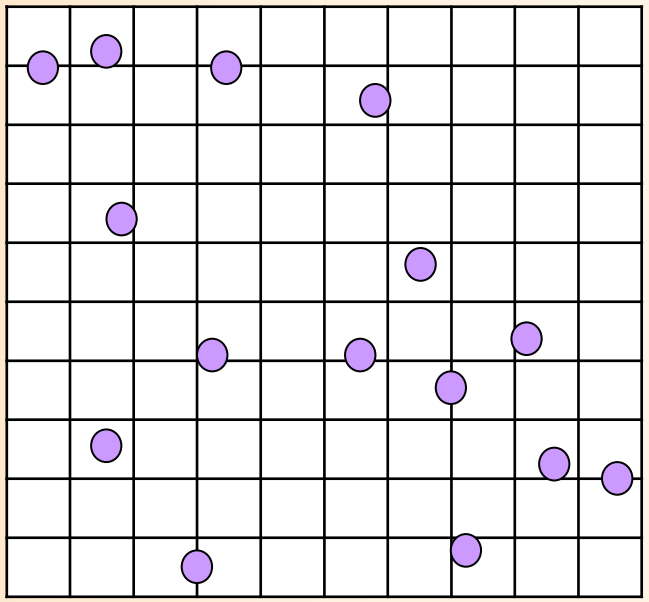
空間２分割木（BSP）

- 空間内の各ポリゴンを含む超平面で、空間を２分割していく
  - ポリゴンが超平面と交差したら、ポリゴンも分割
  - 建物などのインデックスによく使われる
  - 可視範囲の高速判定にも利用
  - 一昔前の一人称シューティング（FPS）ゲームによく用いられた手法



ボクセル

- あらかじめ空間を固定のグリッドに分ける
  - 各グリッドにどのオブジェクトが含まれるかを記録
  - 物体に応じた分割が必要ない
  - ツリーを辿る必要がないので、アクセスが高速
  - 多くのメモリを必要とする
  - グリッドのサイズが重要
  - グリッドの圧縮

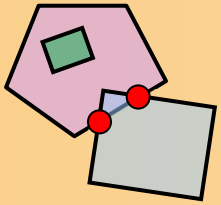


衝突判定

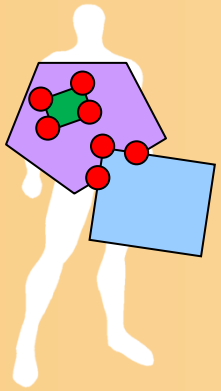
- 近似形状による衝突（交差）判定
- 空間インデックス
- ポリゴンモデル同士の衝突（交差）判定

ポリゴンモデル同士の交差判定

- サーフェス同士として判定する方法
  - ２枚のポリゴン同士の交差判定に帰着できる
    → 線分と面の交差判定の繰り返しにより判定
  - 一方の内部にもう一方が完全に入ると、判定できない

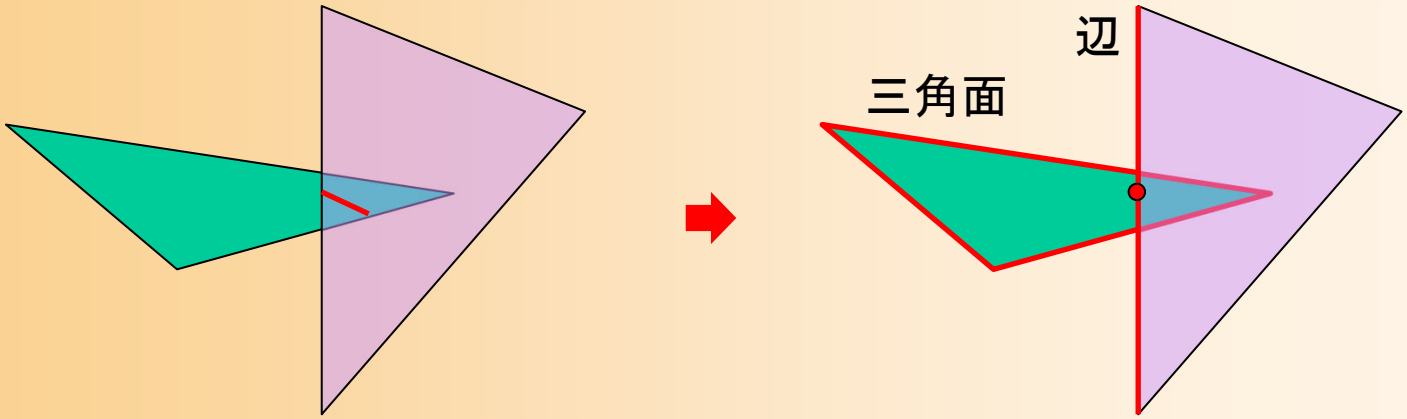


- ソリッドモデル同士として判定する方法
  - 線分と面の交差判定と、凸多面体と点の包含判定の繰り返しにより判定
    - 上のサーフェス同士の交差判定に、包含判定も追加
  - ポリゴン数が多いと判定に時間がかかる



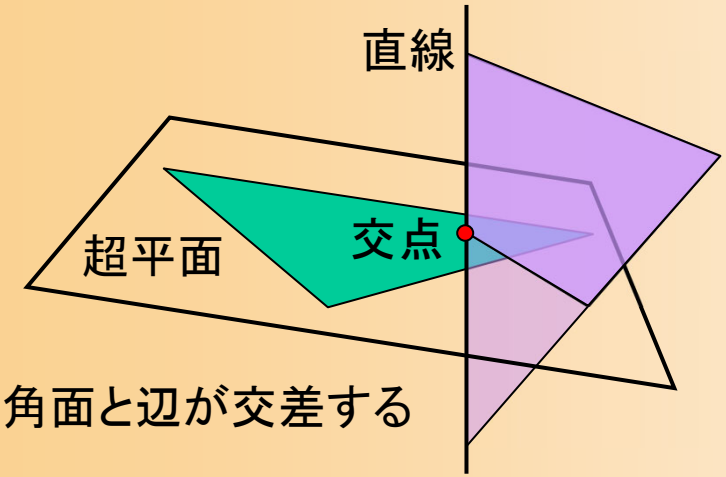
サーフェス同士の交差判定（1）

- ２枚のポリゴン（三角面）同士の交差判定
  → 三角面と辺の交差判定の繰り返しで判定
  - 一方の三角面の辺が、もう一方の三角形と交差すれば、２枚の三角面は交差すると判定できる
  - ２枚の三角面の全ての辺（3本×2枚＝6本）に対して、もう一方の三角形との交差を判定

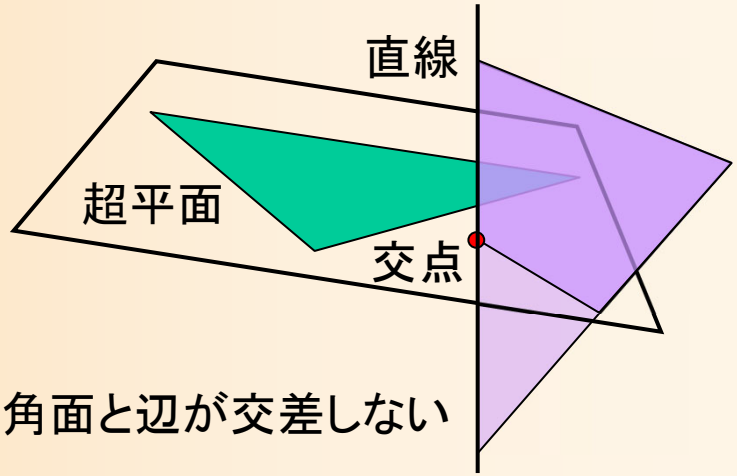


サーフェス同士の交差判定（2）

- 三角面と辺の交差判定 → 超平面と直線の交点から判定
  - 一方の三角面を含む超平面と、もう一方の三角面の辺を含む直線の交点を計算
  - 交点が辺上にあり、かつ三角面の内側にあれば、辺と三角面は交差する



三角面と辺が交差する



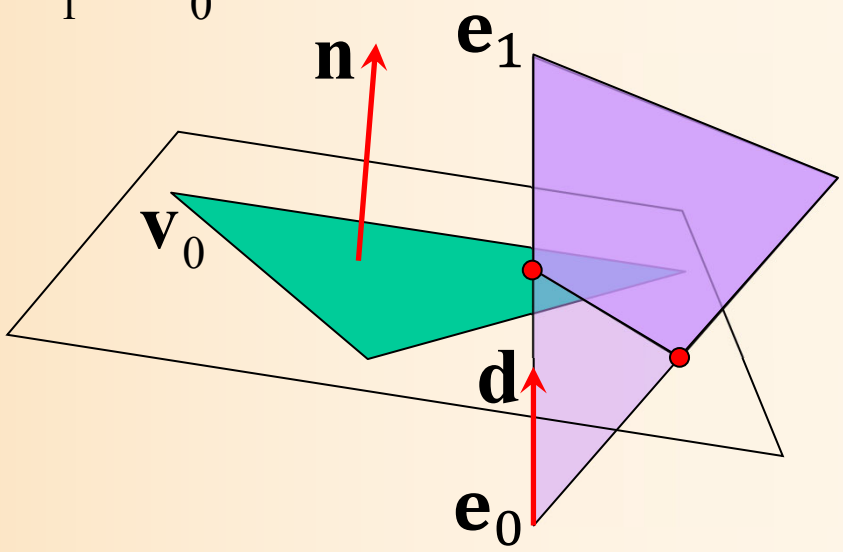
三角面と辺が交差しない

サーフェス同士の交差判定（3）

- 三角面（超平面）と辺（直線）の交点計算
  - 超平面上の点 $\mathbf{p}$ を表す方程式

    $\mathbf{n}(\mathbf{p}-\mathbf{v}_0)=0$ $\mathbf{n}$ は超平面の法線
  - 直線上の点 $\mathbf{p}$ を表す式

    $\mathbf{p}=t\mathbf{d}+\mathbf{e}_0$
  - ２つの式から交点が求まる $\mathbf{d}=\mathbf{e}_1-\mathbf{e}_0$
    - t が 0～1 の範囲のとき、辺と超平面は交差（交点が辺上に存在）



サーフェス同士の交差判定（4）

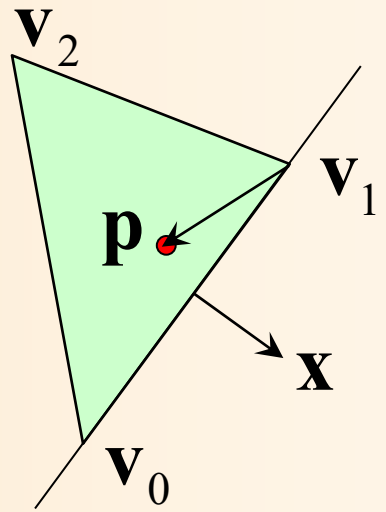
- 交点が三角面の内側にあるかの判定
  - ある辺（$\mathbf{v}_0\mathbf{v}_1$ ）に垂直なベクトル $\mathbf{x}$ を計算

$$\mathbf{x} = (\mathbf{v}_1 - \mathbf{v}_0) \times \mathbf{n}$$

  - 辺から見て、もうひとつの頂点 $\mathbf{v}_2$ と交点 $\mathbf{p}$ が同じ方向にあれば、内側にあると判定

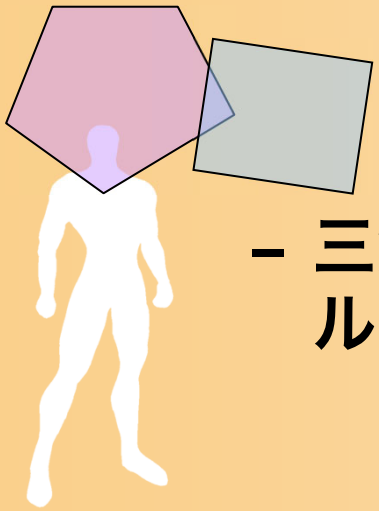
$$(\mathbf{v}_2 - \mathbf{v}_0) \cdot \mathbf{x} \times (\mathbf{p} - \mathbf{v}_0) \cdot \mathbf{x} > 0$$

  - 3つの辺に対して、全て内側にあれば、交点は三角面の内部にあると判定できる



サーフェス同士の交差判定（5）

- 全体の処理のまとめ
- ポリゴンモデル同士の交差判定
  - 各三角面同士の交差判定
    - 三角面と各辺の組の交差判定
      - 三角面を含む超平面と、辺を含む直線の、交点の計算
      - 交点が、辺上にあるかと、三角面内にあるかを判定
        - » 交点が、三角面の全ての辺の内側にあるかを判定
          - » 1組でも三角面と辺が交差する場合は、ポリゴンモデル同士は交差すると判定して、処理を終了



  - 三角面同士の交差が一つもなければ、ポリゴンモデル同士は交差しないと判定

プログラムの作成（1）

- 幾何形状モデル同士の交差判定

```
//  幾何形状モデル同士の交差判定
bool  CheckCollision(
    const Geometry * g0, const Geometry * g1,
    const Matrix4f & f0, const Matrix4f & f1,
    Point3f & cross_point )
{
    // どこか一箇所でも交差する場合は、戻り値として true を返す
    // 交差しない場合は、false を返す
    // 交差する場合は、一つの交点位置を cross_point に格納して返す

    // 片方の幾何形状モデルの各辺と、もう一方の幾何形状モデルの
    // 各面の交差判定（三角面と辺の交差判定）を繰り返すことにより判定
    // 一つでも交差する組み合わせがあれば、形状モデル同士も交差する
}
```

入力（２つの幾何形状モデル＋それぞれの位置・向きを表す変換行列）

出力（交差の有無）

出力（交差する場合の交点の情報）

プログラムの作成（2）

- 三角面と辺（線分）の交差判定

```
//  三角面と線分の交差判定
bool  CheckCross(
    const Point3f & f0, const Point3f & f1, const Point3f & f2,
    const Point3f & s0, const Point3f & s0,
    Point3f & cross_point )
{
    // 三角面と直線が交差する場合は、戻り値として true を返す
    // 交差しない場合は、false を返す
    // また、交差する場合は、交点の位置を cross_point に格納して返す
}
```

入力（三角形と線分の情報）

出力（交差の有無）

出力（交差する場合の交点の情報）

プログラムの作成（3）

```
//  三角面と線分の交差判定
bool  CheckCross(
    const Point3f & f0, const Point3f & f1, const Point3f & f2,
    const Point3f & s0, const Point3f & s0,
    Point3f & cross_point )
{
    // 三角面を含む超平面の法線ベクトルを計算
    // 線分を含む直線の方向ベクトルを計算

    // 超平面と直線の交点 p を計算
    // 交点が線分の範囲にない場合（t < 0 or t > 1）や、
    // 超平面と直線が並行の場合（t = ±∞）は、交差しないと判定

    // 交点が三角形の内部にあるかを判定
    // 三角形の全ての辺に対して、交点が辺の内側にあるかを判定
    // 判定結果を返す、交点位置を cross_point に格納して返す
}
```

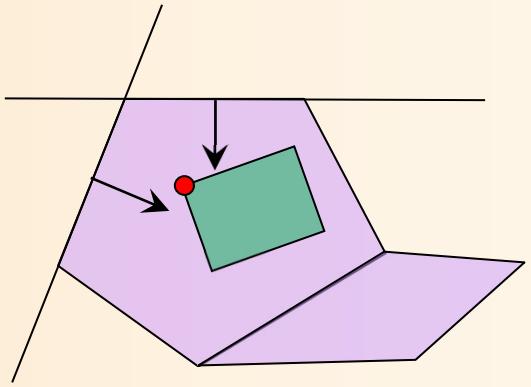
$$\mathbf{n}\left(\mathbf{p}-\mathbf{v}_0\right)=0$$

$$\mathbf{p}=t\mathbf{d}+\mathbf{e}_0$$

ソリッドモデル同士の交差判定

1. ポリゴンモデルと点の包含判定

- ポリゴンモデルを複数の凸多面体に分解
- 凸多面体の全ての面の内側にあれば包含
  - 各面ごとに、面を含む超平面を方程式で表す
  - 超平面の式に点座標を代入し、符号を判定

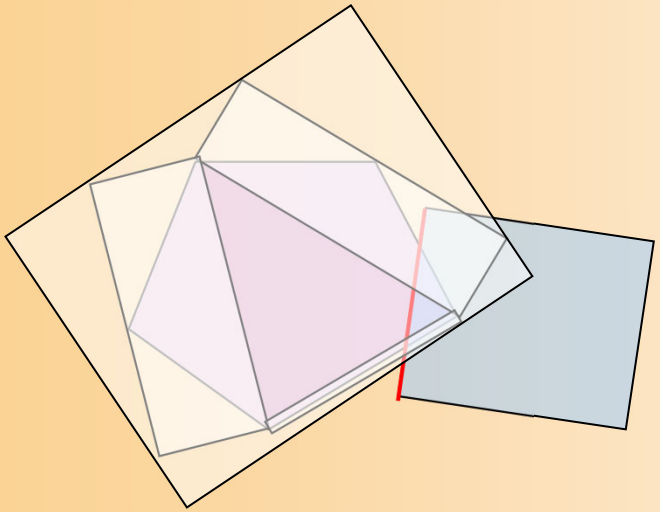


2. 三角面同士の交差判定

- サーフェスモデル同士の交差判定と同じ

交差判定の効率化

- ポリゴンモデル内にツリーを導入し、ポリゴンを階層的に管理することで、効率化できる
  - どのポリゴン同士の判定を行うべきかを効率的に決定



交差判定の種類

- 単純に交差しているかどうかだけ判定
  - １つでも交差が見つかればそこで終了
- 交点を全て計算
  - 衝突の処理のために必要
- めり込みの位置や深さを計算
  - 接触（めり込み回避）の処理のために必要

- アプリケーションが必要とする情報によって、必要な交差判定の種類は異なる

衝突判定

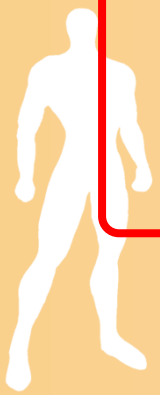
- 近似形状による衝突（交差）判定
- 空間インデックス
- ポリゴンモデル同士の衝突（交差）判定

衝突判定（交差判定）の流れ

1. 空間インデックスによる交差候補の探索
   - まずは、交差している可能性があるポリゴンモデルの組を、空間インデックスを使って求める
2. 近似形状による交差判定
   - 交差する可能性のあるポリゴンモデルが、交差するかどうか、近似形状を使って判定
3. ポリゴンモデル同士の交差判定
   - 最終的には、ポリゴンモデルをもとに交差判定
     - モデル内部のインデックスにより交差する可能性がある三角面を求める → 三角面同士の交差判定

今日の内容

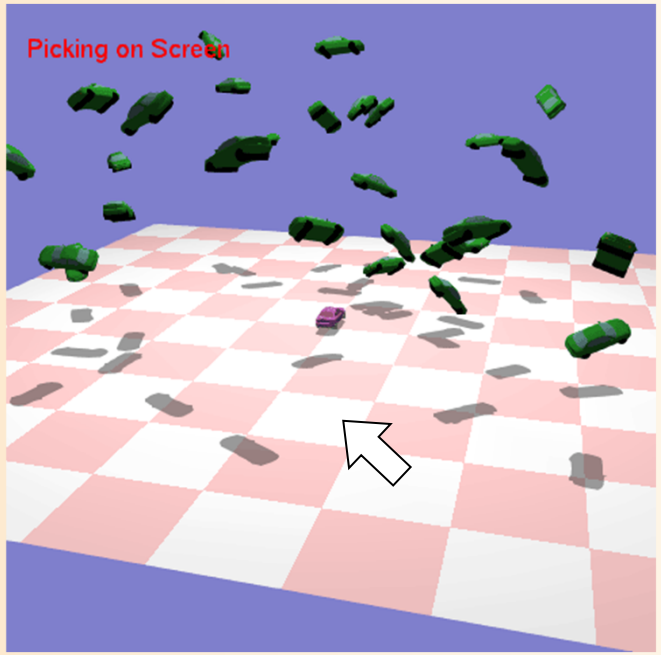
- 衝突判定
  - 近似形状による衝突判定
  - 空間インデックス
  - ポリゴンモデル同士の衝突判定
- ピッキング
  - サンプルプログラム
  - スクリーン座標系での判定
  - ワールド座標系での判定
  - レポート課題



ピッキング

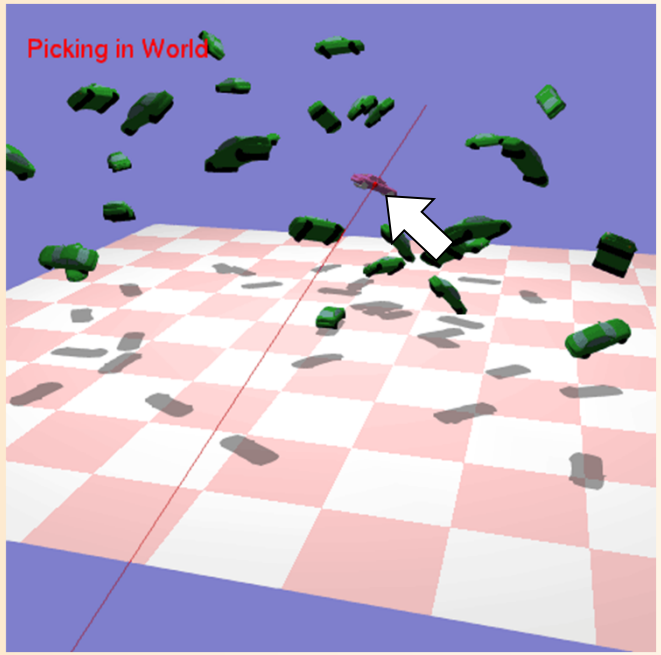
ピッキング

- 画面上で３次元空間の物体を選択する操作
  - 画面上の指定した点に表示されているオブジェクトを判定する処理
  - 多くのアプリケーションで必要となる技術
  - 場合によっては、選択点の座標まで求める必要がある
  - 幾何学的な計算が必要



デモプログラム

- ピッキング
  - マウスの左クリックで、オブジェクトを選択
    - 選択されたオブジェクトの色を変更
  - 任意の視点から選択可能
  - ピッキングの判定方法
    - ワールド座標系での判定
    - スクリーン座標系での判定
  - 選択点の表示
    - ワールド座標系での判定時
    - 視線ベクトルも表示可能




Picking
Picking on Screen
ピッキング
Picking

サンプルプログラム

- デモプログラムの一部のサンプルプログラム（picking_sample.cpp）
  - 幾何形状モデルの読み込み
  - 指定された数（30個）のオブジェクトの位置・向きをランダムに初期化
  - マウスの左ボタンがクリックされたら、マウス位置にもとづいて、ピッキングの処理を呼び出し
- 幾何形状の読み込み・描画関数（obj.h/cpp）
  - 過去の授業で扱った内容、影の描画も含む
- vecmath 補助関数（vecmath_gl.h）

幾何形状の読み込み・描画関数

```
// 幾何形状データ(Obj形式用)
struct  Obj
      定義は省略(第4回目の講義資料を参照)

// Objファイルの読み込み
Obj *  LoadObj( const char * filename );

// Mtlファイルの読み込み
void  LoadMtl( const char * filename, Obj * obj );

//幾何形状モデルのスケーリング(スケーリング後のサイズを返す)
void  ScaleObj( Obj * obj, float max_size, ・・・ );

//幾何形状モデル(Obj形状)の描画
void  RenderObj( Obj * obj );

//幾何形状モデル(Obj形状)の影の描画(ポリゴン投影による影の描画)
void  RenderShadow( Obj * obj, float matrix[ 16 ], ・・・ );
```

影の描画については後日の講義で扱う（今回は空欄のままでも良い）

オブジェクトの幾何形状・配置情報

```
// 表示用の幾何形状モデル
Obj *  object;

// オブジェクトの配置情報
struct  ObjectInfo
{
    // 位置・向き
    Point3f   pos;
    Matrix3f  ori;
    // 変換行列（位置・向きから描画用に計算）
    Matrix4f  frame;
    // 画面上での位置（ピッキングのために計算）
    Point2f   screen_pos;
};

// 全オブジェクトの配列
int  num_objects = 0;
ObjectInfo *  objects = NULL;
```

オブジェクトの中心（幾何形状モデルの原点）の位置・向き

num_objects 分の配列の先頭アドレス

ピッキング処理に関する情報

```
// ピッキング判定方法を表す列挙型
enum  PickModeEnum
{
    PICK_SCREEN,
    PICK_WORLD,
};
// ピッキング判定方法
PickModeEnum  pick_mode = PICK_SCREEN;

// オブジェクトの選択情報（選択中のオブジェクト番号）
int  selected_object_no = -1;

//  マウスクリック時に呼ばれるコールバック関数
void  MouseClickCallback( int button, int state, int mx, int my )
{
    // 左ボタンが押されたら、オブジェクトを選択（ピッキング処理）
    if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_DOWN ) )
        selected_object_no = PickObject( mx, my );
    ...
```

２つの判定方法のどちらを使用するかを表す

未選択時には －1 を格納

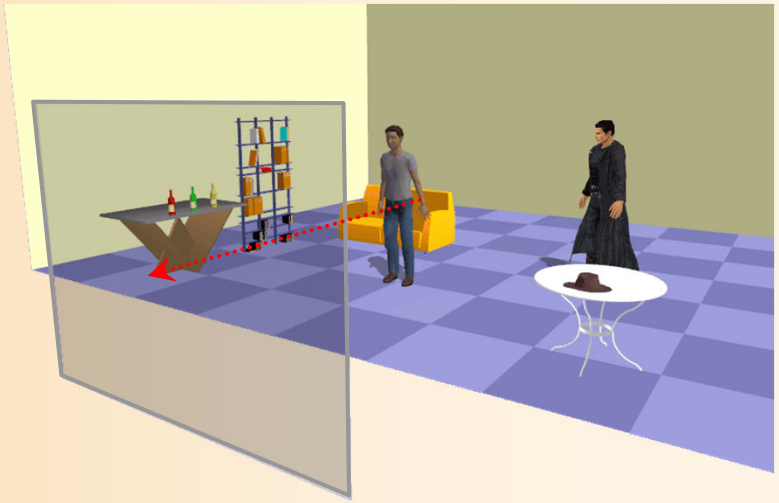
ピッキング処理を行って結果を記録

ピッキングの実現方法

- 単純には実現できないため、工夫が必要となる
- ピッキングの問題点
  - 画面上の座標と３次元空間の座標は異なる
  - 複数の物体が選択される可能性がある
- これらの問題点を解決するための実現方法
  - 方法１：スクリーン座標系での判定
  - 方法２：ワールド座標系での判定

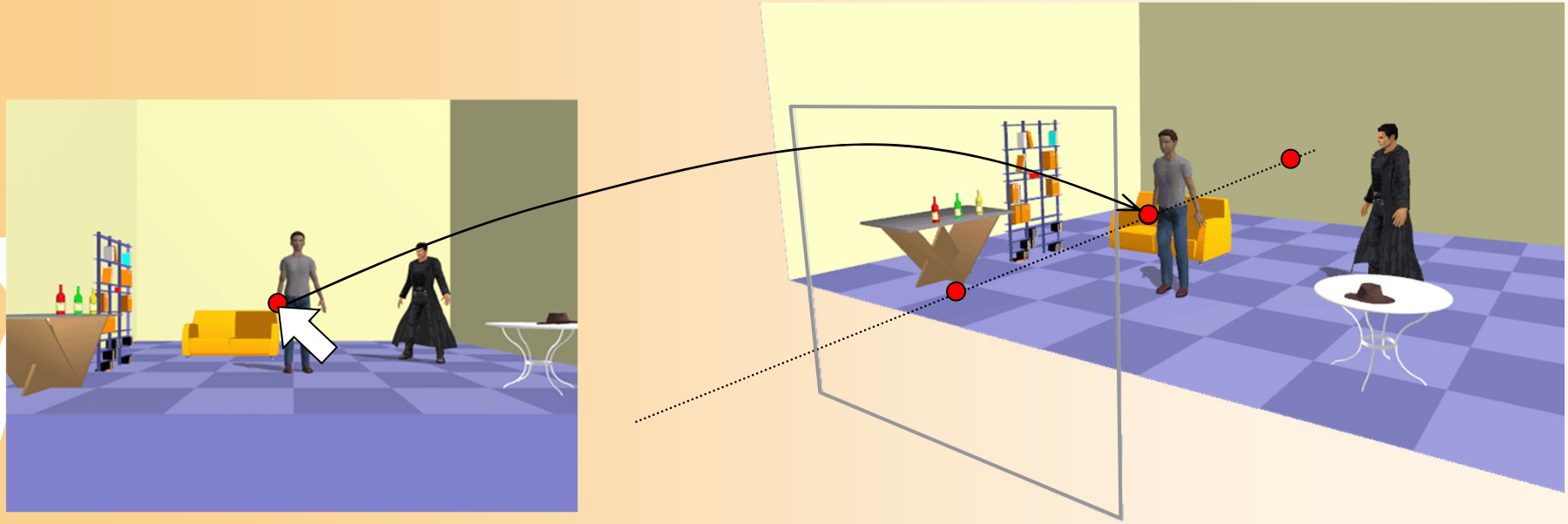
ピッキングの問題点（1）

- 画面上の座標と３次元空間の座標は異なる
  - 仮想空間の座標は、視野変換行列、透視射影行列によって、画面に投影される
  - 画面上のマウス位置（スクリーン座標系）と、物体の位置（ワールド座標系）は、そのままでは比較できない



ピッキングの問題点（2）

- 画面上の点は、仮想空間内の半直線に相当
  - 画面上の指定された点が、複数の物体に対応する可能性がある
  - 何らかの条件により、選択された物体を決定する必要がある



ピッキングの判定方法

- 方法１：スクリーン座標系で判定する
  - 各オブジェクトの画面上での位置を計算
    - ワールド座標系でのオブジェクトの位置（中心位置）をスクリーン座標系に投影
  - マウス位置と各オブジェクトの距離により判定
- 方法２：ワールド座標系で判定する
  - マウス位置に相当するワールド座標系での半直線を求める
  - 各オブジェクトを構成するポリゴンと半直線の交差判定により判定

ピッキングの関数の定義（1）

- 方法１：スクリーン座標系で判定する

```
// ピッキング処理（スクリーン座標系）
int  PickObjectScreen( int mouse_x, int mouxe_y );
    // 選択されたオブジェクトのオブジェクト番号を返す
    // どのオブジェクトも選択されていない場合は、－1 を返す
```

クリック時のマウス座標を引数として渡す

- 方法２：ワールド座標系で判定する

```
// ピッキング処理（ワールド座標系）
int  PickObjectWorld( int mouse_x, int mouse_y );
    // 選択されたオブジェクトのオブジェクト番号を返す
    // どのオブジェクトも選択されていない場合は、－1 を返す
```

クリック時のマウス座標を引数として渡す

ピッキングの関数の定義（2）

- 現在の使用方法に応じた関数を呼び出し

```
// ピッキング処理
int  PickObject( int mouse_x, int mouse_y )
{
        // スクリーン座標系で判定
        if ( pick_mode == PICK_SCREEN )
                return  PickObjectScreen( mouse_x, mouse_y );

        // ワールド座標系で判定
        else if ( pick_mode == PICK_WORLD )
                return  PickObjectWorld( mouse_x, mouse_y );

        return  -1;
}
```

ピッキングの判定方法

- 方法１：スクリーン座標系で判定する
  - 各オブジェクトの画面上での位置を計算
    - ワールド座標系でのオブジェクトの位置（中心位置）をスクリーン座標系に投影
  - マウス位置と各オブジェクトの距離により判定
- 方法２：ワールド座標系で判定する
  - マウス位置に相当するワールド座標系での半直線を求める
  - 各オブジェクトを構成するポリゴンと半直線の交差判定により判定

方法１：スクリーン座標系で判定

1. 全てのオブジェクトの画面上の位置を計算
   - ワールド座標系の位置（wx,wy,wz） を
     スクリーン座標系の位置（sx,sy） に変換
   - オブジェクトの中心位置を使用
2. クリック時のマウスの位置と、各オブジェクトの距離をもとに、選択オブジェクトを判定
   - 両者のスクリーン座標系での距離を計算
   - 距離が最も近いオブジェクトを選択
   - 閾値以内のオブジェクトがなければ選択なし

スクリーン座標系への変換

- ワールド座標系の位置（wx,wy,wz） を スクリーン座標系の位置（sx,sy） に変換
  - gluProject（）関数により計算できる
    - レンダリング・パイプライと同様の座標変換の計算を行い、画面上での座標を計算する関数
    - OpenGLに設定されている変換行列・ビューポートの値を取得して、引数として渡す必要がある
  - 自分で行列計算とビューポート変換を行っても、構わない（同じ計算結果になる）

スクリーン座標系への変換

- ワールド座標系の位置（wx,wy,wz） を
  スクリーン座標系の位置（sx,sy） に変換

```
// OpenGL の変換行列を取得
double model_view_matrix[ 16 ], proj_mat[ 16 ], px,py,pz;
int  viewport_param[ 4 ];
glGetDoublev( GL_MODELVIEW_MATRIX, model_view_matrix );
glGetDoublev( GL_PROJECTION_MATRIX, projection_matrix );
glGetIntegerv( GL_VIEWPORT, viewport_param );

// ワールド座標の点をスクリーン座標に投影
gluProject( wx, wy, wz, model_view_matrix, projection_matrix,
    viewport_param, &px, &py, &pz );
sx = px;
sy = viewport_param[ 3 ] - py;  // 左上が 0 になるように変換
```

全オブジェクトの画面上の位置を計算

```
//  全オブジェクトの画面上の位置を更新
void  UpdateObjectProjection()
{
    // ワールド→カメラ座標系への変換行列が設定されているとする

    // OpenGL の変換行列を取得
    ・・・

    // 各オブジェクトの画面上の位置を計算
    for ( int i=0; i<num_objects; i++ )
    {
        // i番目のオブジェクトの情報を取得
        ObjectInfo *  obj = &objects[ i ];

        // ワールド座標の点をスクリーン座標に投影
        ・・・
        obj->screen_pos.x = ???;    obj->screen_pos.y = ???;
```

前スライドの説明通りの処理

オブジェクトの選択処理

```
//  オブジェクト選択処理（スクリーン座標系）
int  PickObjectScreen( int mouse_x, int mouxe_y )
{
    // 選択されたかどうかを判定するための、画面上での距離の閾値
    const float  threshold = 20.0f;

    // 全オブジェクトの画面上の位置を更新
    UpdateObjectProjection();

    // 各オブジェクトの位置とマウス位置の距離を計算
    for ( int i=0; i<num_objects; i++ )
    {
        ObjectInfo *  obj = &objects[ i ];
        ・・・・
    }
    return   ・・・;
}
```

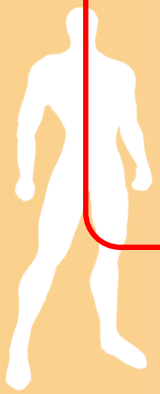
前スライドの関数を呼び出し

画面上での距離が最小の
オブジェクトを選択
（距離が閾値以下のもののみ）

選択オブジェクト番号を返す（閾値以下の
距離のオブジェクトがなければ －1 を返す）

ピッキングの判定方法

- 方法１：スクリーン座標系で判定する
  - 各オブジェクトの画面上での位置を計算
    - ワールド座標系でのオブジェクトの位置（中心位置）をスクリーン座標系に投影
  - マウス位置と各オブジェクトの距離により判定
- 方法２：ワールド座標系で判定する
  - マウス位置に相当するワールド座標系での半直線を求める
  - 各オブジェクトを構成するポリゴンと半直線の交差判定により判定

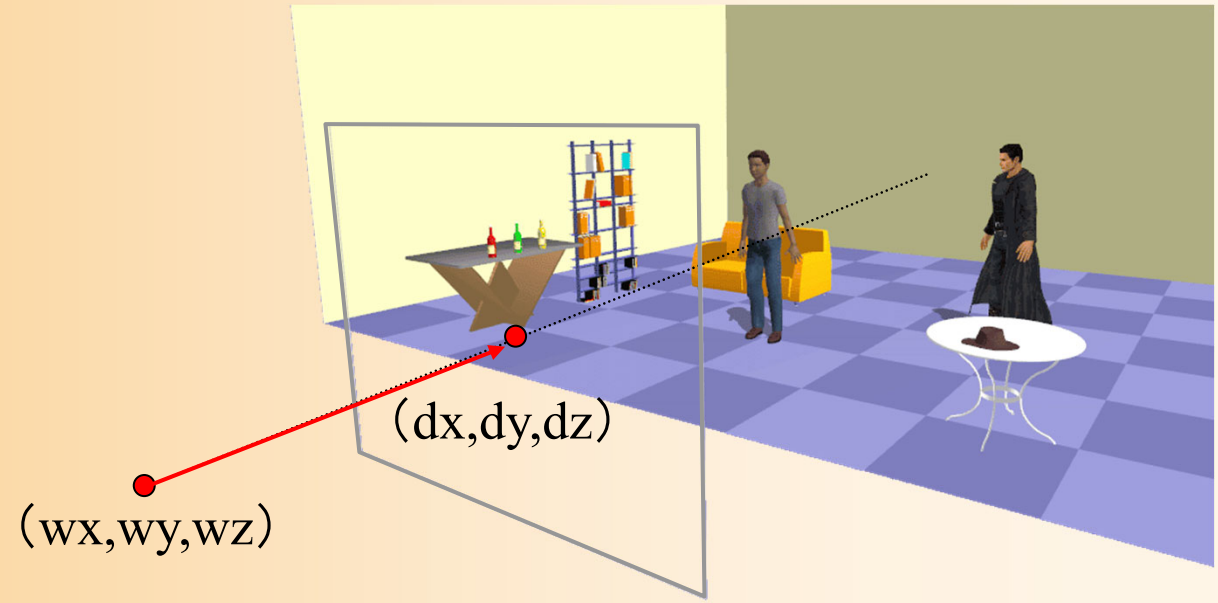


方法２：ワールド座標系で判定

1. マウス位置に対応する３次元空間の半直線を求める
   - マウス座標に対応する半直線を表す、視点位置（wx,wy,wz）と 視線ベクトル（dx,dy,dz）を求める
2. 半直線と交差するオブジェクトを求める
   - 半直線と、各オブジェクトを構成する各三角面の交差判定と交点計算
   - 最も視点に近い交点とオブジェクトを求める

方法２：ワールド座標系で判定

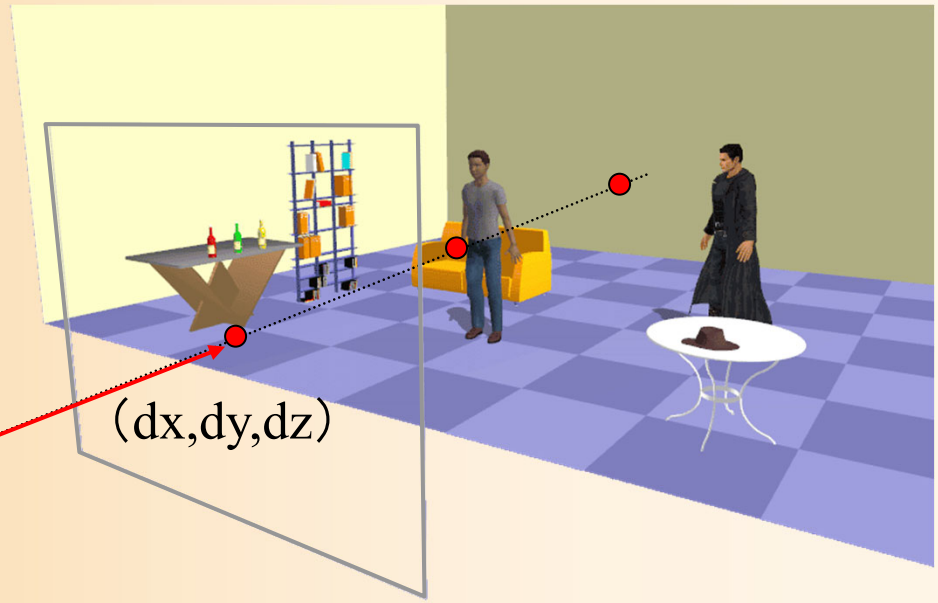
1. マウス位置に対応する３次元空間の半直線を求める
   - マウス座標に対応する半直線を表す、視点位置（wx,wy,wz）と 視線ベクトル（dx,dy,dz）を求める



方法２：ワールド座標系で判定

2. 半直線と交差するオブジェクトを求める

- 半直線と、各オブジェクトを構成する三角面の交差判定と交点計算
- 最も視点に近い交点とオブジェクトを求める



マウス位置に対応する半直線の計算

- マウス位置に対応する半直線を表す、視点位置（wx,wy,wz）と 視線ベクトル（dx,dy,dz）を求める
- スクリーン座標系からワールド座標系へ変換
  - カメラ座標系の原点をワールド座標系に変換することで、視点位置（wx,wy,wz）が求まる
  - カメラ座標系でのスクリーン上（手前のクリップ面）のマウス位置（sx,sy）をワールド座標に変換して視点位置との差を計算することで、視線ベクトル（dx,dy,dz）が求まる

ワールド座標系への変換

- gluUnProject 関数により、スクリーン座標系からワールド座標系の変換を計算できる
  - gluProject 関数の逆変換を計算する関数
  - 自分で同様の計算を行っても良い
- gluUnProject 関数の引数（x,y,z）には、スクリーン座標系での３次元位置を指定
  - z値には、射影変換後の奥行値（の逆数）を設定
    - 視点位置では、z値は無限大
      - 適当な大きな値を使用（例：10000000000.0）
    - スクリーン上（手前のクリップ面）では、z値は 1.0

マウス位置に対応する半直線の計算

- 視点位置（wx,wy,wz）と 視線ベクトル（dx,dy,dz）を計算

```
// OpenGL の変換行列を取得
double model_view_matrix[ 16 ], proj_mat[ 16 ];
int  viewport_param[ 4 ];
glGetDoublev( GL_MODELVIEW_MATRIX, model_view_matrix );
glGetDoublev( GL_PROJECTION_MATRIX, projection_matrix );
glGetIntegerv( GL_VIEWPORT, viewport_param );

// スクリーン座標の点をワールド座標の半直線に変換
gluUnProject( 0.0, 0.0, 10000000000.0, model_view_matrix,
        projection_matrix, viewport_param, &wx, &wy, &wz );
gluUnProject( sx, viewport_param[ 3 ] - sy, 1.0, model_view_matrix,
        projection_matrix, viewport_param, &dx, &dy, &dz );
dx = dx – wx;    dy = dy – wy;    dz = dz – wz;
```

視点位置

視線ベクトル

半直線と交差するオブジェクトの探索

```
//  オブジェクト選択処理（ワールド座標系）
int  PickObjectWorld ( int mouse_x, int mouxe_y )
{
    // マウス位置に対応する3次元空間の直線を求める
    Point3f  line_org( wx, wy, wz );    Vector3f  line_vec( dx, dy, dz );
    ・・・

    // 各オブジェクトと直線の交差判定
    for ( int i=0; i<num_objects; i++ )
    {
        const ObjectInfo &  obj = objects[ i ];

        // オブジェクトの各ポリゴンとの交差判定
        for ( int j=0; j<object->num_triangles; j++ )
        {
            ・・・
            bool  cross = CheckCross( ・・・ );
```

前スライドの方法により計算

半直線とポリゴン（三角面）の交差判定

半直線と三角面の交差判定

- 衝突判定での説明・計算式をもとに作成
  - 三角面の３点の座標、半直線の点と方向ベクトルを入力として受け取り、交差判定結果を返す

入力（三角形と半直線の情報）

出力（交差の有無）

出力（交差する場合の交点の情報）

```
//  三角面と半直線の交差判定
bool  CheckCross(
    const Point3f & p0, const Point3f & p1, const Point3f & p2,
    const Point3f & seg_org, const Vector3f & seg_vec,
    Point3f & cross_point )
{
    // 三角面と直線が交差する場合は、戻り値として true を返す
    // 交差しない場合は、false を返す
    // また、交差する場合は、交点の位置を cross_point に格納して返す
}
```

半直線と三角面の交差判定

```
//  三角面と半直線の交差判定
bool  CheckCross(
    const Point3f & p0, const Point3f & p1, const Point3f & p2,
    const Point3f & seg_org, const Vector3f & seg_vec,
    Point3f & cross_point )
{
    // 三角形を含む超平面の法線ベクトルを計算
    // 半直線の方向ベクトルは引数で与えられる

    // 超平面と直線の交点 p を計算
    // 交点が線分の範囲にない場合（t < 0）や、
    // 超平面と直線が並行の場合（t = ±∞）は、交差しないと判定

    // 交点が三角形の内部にあるかを判定
    // 三角形の全ての辺に対して、交点が辺の内側にあるかを判定
    // 判定結果を返す、交点位置を cross_point に格納して返す
}
```

$$\mathbf{n}\left(\mathbf{p}-\mathbf{v}_0\right)=0$$

$$\mathbf{p}=t\mathbf{d}+\mathbf{e}_0=0$$

半直線と三角面の交差判定

三角面と線分の交差判定と同様の処理

```
//  三角面と半直線の交差判定
bool  CheckCross(
    const Point3f & p0, const Point3f & p1, const Point3f & p2,
    const Point3f & seg_org, const Vector3f & seg_vec,
    Point3f & cross_point )
{
    // 三角形を含む超平面の法線ベクトルを計算
    // 半直線の方向ベクトルは引数で与えられる

    // 超平面と直線の交点 p を計算
    // 交点が線分の範囲にない場合（t < 0）や、
    // 超平面と直線が並行の場合（t = ±∞）は、交差しないと判定

    // 交点が三角形の内
    // 三角形の全ての辺に
    // 判定結果を返す、交点
}
```

$$\mathbf{n}(\mathbf{p}-\mathbf{v}_0)=0$$

$$\mathbf{p}=t\mathbf{d}+\mathbf{e}_0=0$$

半直線なので $t > 1$ の場合も交差

判定のための計算を実装
vecmath での各種ベクトル演算の方法は、
vecmath のドキュメントを参照
内積計算（dot関数）や外積計算（cross関数）が利用可能

判定方法の比較

- 方法１：スクリーン座標系で判定する
  - 処理は単純
  - 点単位でしか判定できない（大きさを考慮できない）
  - 複数の候補があるときの決定が困難（3次元空間での前後関係や距離を考慮することが困難）
  - 全物体（選択可能点）に対して座標変換を行う必要があるので、物体数が多いと処理が遅くなる
- 方法２：ワールド座標系で判定する
  - 処理はやや複雑
  - 正確な選択点を求めることが可能
  - 全物体（選択可能面）と判定を行うと処理が遅くなる
  - 効率的に処理すれば、方法1よりも高速化が可能

参考：スクリーン座標系での判定の別方法

- OpenGLのセレクション機能を使った判定方法（画面内の指定範囲に描画されたポリゴンの情報を取得する機能）
  1. glSelectBuffer関数で、セレクションの結果を格納する領域を確保
  2. glRenderMode(GL_SELECT) で、セレクションモードを有効にする
  3. gluPickMatrix関数で、マウス座標の周辺領域のみを描画可能領域に指定
  4. glLoadName関数で、ポリゴンに番号を付けて、ポリゴンの描画処理を行う（実際は描画されない）
  5. 描画可能領域に描画されたポリゴンの番号を取得

まとめ

- 衝突判定
  - 近似形状による衝突判定
  - 空間インデックス
  - ポリゴンモデル同士の衝突判定
- ピッキング
  - サンプルプログラム
  - スクリーン座標系での判定
  - ワールド座標系での判定
  - レポート課題

レポート課題

- ピッキングを実現するプログラムを作成
  1. スクリーン座標系でのピッキング
  2. ワールド座標系でのピッキング
     - 三角面と半直線の交差判定
  - サンプルプログラム（picking_sample.cpp）をもとに作成したプログラムを提出
    - 他の変更なしのソースファイルやデータは、提出する必要はない
  - Moodleの本講義のコースから提出
  - 締切： Moodleの提出ページを参照

レポート課題 提出方法

Moodleから、以下の２つのファイルを提出

- 作成したプログラム（テキスト形式）
  - picking_sample.cpp
- 変更箇所のみを抜き出したレポート（PDF）
  - Moodle に公開している LaTeX のテンプレートをもとに、作成する
    - これまでのレポートと同様

レポート課題 演習問題

- レポート課題の提出に加えて、レポート課題の理解度を確認するための Moodle 演習問題にも解答する
  - 解答締切は、レポート提出と同じ
  - レポート課題のヒントにもなっているので、レポート課題で分からない箇所があれば、演習問題の説明・選択肢を参考にして考えても良い
  - 締切後に解答が表示されるので確認する
    - レポート課題では、正しく動作するプログラムが提出されていれば、演習問題の正答の通りのプログラムが作成されていなくとも構わない

次回予告

- キャラクタ・アニメーション（1）
  - 人体モデルの基礎
  - 人体モデル（骨格・姿勢・動作）の表現
    - 骨格モデルの表現
    - 姿勢・動作の表現
    - 形状モデルの表現
  - 人体モデルの作成
  - 動作データの作成
    - キーフレームアニメーション
    - モーションキャプチャ